Finding Insecure State Dependency in DApps via Multi-Source Tracing and Semantic Enrichment

Jingwen Zhang[†§], Yuhong Nan[†*], Wei Li[†], Kaiwen Ning[†§], Zewei Lin[†§], Zitong Yao[†],
Yuming Feng[§], Weizhe Zhang[‡§], and Zibin Zheng[†],
[†]Sun Yat-sen University, {zhangjw273, liwei378, ningkw, linzw3, yaozt}@mail2.sysu.edu.cn, {nanyh,zhzibin}@mail.sysu.edu.cn
[‡]Harbin Institute of Technology, wzzhang@hit.edu.cn,
[§]Peng Cheng Laboratory, fengym@pcl.ac.cn

***Abstract*—Decentralized Applications (DApps) serve as the gateway to utilizing blockchain technology. As their prevalence continues to grow, DApps are becoming increasingly interconnected. For instance, a DApp does not need to manage the prices of various tokens internally, as it can retrieve this information from other DApps that provide more up-to-date data. However, such deep reliance also introduces more attack surfaces, posing greater risks to both DApps and their users. In this paper, we refer to the security threat arising from the interdependence of DApps as Insecure State Dependency (ISD). Public reports indicate that ISD has led to losses exceeding 340 million USD.**

Existing ISDs are mostly found by extensive manual auditing and lucky incidents, as automated discovery of such issues is extremely difficult. More specifically, it is by no means trivial to (1) achieve precise data tracking in the intertwined and invisible interactions of DApps, (2) obtain fine-grained semantic information in low semantic bytecode. In this paper, we propose a novel framework, called *InsFinder*, for detecting ISD in DApps. Specifically, *InsFinder* consists of three unique modules to overcome the aforementioned challenges. (1) *InsFinder* employs dynamic cross-DApp taint analysis to achieve accurate multi-source data tracking in heavily coupled DApp interactions. (2) *InsFinder* uses source mapping to map bytecode identifiers into meaningful source code, such as variable names or statements, enabling a deeper understanding of bytecode. (3) *InsFinder* implements fine-grained access control and static analysis for ISD entry point detection. Evaluation on a manually annotated dataset with 93 real-world ISDs shows that *InsFinder* successfully detects 72 of them, achieving a precision of 84.7% and a recall of 77.4%. Furthermore, *InsFinder* successfully uncovers 165 previously unreported ISDs across 122 DApp projects. These ISDs collectively impact over 2 million USD.

***Index Terms*—Smart Contract, Dynamic Taint Analysis, Source Mapping, Vulnerability Detection**

I. Introduction

As one of the most important applications of blockchain, Decentralized Applications (DApps) [1] utilize blockchain for data storage while processing data through smart contracts – a piece of code that can automatically execute functions and update data. To reduce development costs and enrich the functionalities of DApps, such as token swaps [2], games [3], and lending [4], developers often integrate data or modules from other DApps. However, such deep integration and interaction pose new security threats. For example, if a DApp relies on an external DApp's state for token swap, an attacker can indirectly attack the DApp by manipulating the state of the external DApp, such as the token amount.

* Yuhong Nan is the corresponding author.

Insecure State Dependency. Insecure State Dependency (ISD) is a security threat that arises when a DApp relies on external states but lacks effective sterilization on these states. Here, the `external states` refer to root dirty data, such as the states of other DApps and the external transaction parameters. As shown in Fig. 1, to exploit ISD, an attacker can first invoke `entry function` (the entry point) in one DApp (DApp B) to manipulate the value of state, such as `price`. Then, the attacker targets `victim function` (the victim point) in another DApp (DApp A) that depends on this state, such as calculating the token value based on `price`. Ultimately, the attacker can obtain significantly more funds than expected. Our preliminary investigation reveals that ISD-related incidents, such as price manipulation [5], read-only reentrancy [6], and business logic flaws [7], have already caused losses exceeding 340 million USD until April 2025.

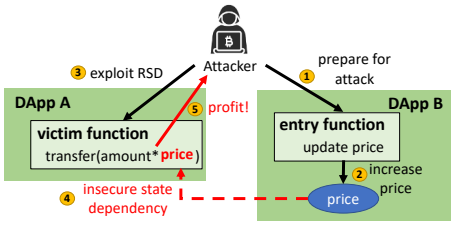


Fig. 1: An example attack that exploits Insecure State Dependency (ISD) across two DApps.

Challenges in detecting ISD. The key to detecting ISDs is determining whether checks on external states are effective, as this can make the state meet expectations. Despite the availability of numerous tools [8], [9], [10], [11] for detecting vulnerabilities in DApps, the detection of ISD is by no means trivial. In detail, the characteristics of ISD present the following unique challenges.

- **Finding checks of critical external states.** Specifying all checks on external states is challenging, as DApp can flexibly conduct checks on these states, like transforming to another variable. Existing methods [8] cannot handle the

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00129

complex form of data interaction, such as cross-variable, cross-contract, and cross-storage data dependencies, leading to the omission of critical external states and corresponding checks during analysis. Besides, not all external states will lead to ISD. Existing research [9], [12] cannot accurately determine whether external states are related to ISD, resulting in the introduction of irrelevant checks.

• **Identifying security checks of ISD.** Not all checks are used to sanitize states, as they can serve diverse purposes based on execution context. Existing pattern-matching approaches [10], [11] fail to comprehend the intention behind various checks in the complicated cross-DApp interactions. Alternatively, Large Language Models (LLMs) show a promising direction, as they excel in code understanding [13], [14]. Nevertheless, since smart contracts are executed based on bytecode and there is a semantic gap between bytecode and source code, LLMs still cannot fully reason the bytecode.

• **Verifying the effectiveness of checks.** Detecting ISD requires specific execution contexts, like execution paths and data sources, to assess the validity of checks, as different contexts can lead to different detecting results. However, obtaining this information is rather difficult. Static analysis, such as Slither [15], cannot reconstruct the complete execution context. Besides, fuzzing methods, like sFuzz [16], struggle to test multiple functions simultaneously.

Our work. In this paper, we introduce *InsFinder*, a novel framework for finding Insecure State Dependency in DApps. Specifically, *InsFinder* receives the DApp contract addresses as input, and outputs the ISD detection results. To address the challenges mentioned above, *InsFinder* proposes the following three unique designs: (1) multi-source tracing to find check bytecode of critical external states, (2) recovering bytecode semantic information, such as variable names and statements, to identify ISD-related checks, (3) fine-grained ISD entry point analysis to verify the effectiveness of security checks in focus.

In detail, to find check bytecode of critical external states, *InsFinder* utilizes dynamic cross-DApp taint analysis to obtain more accurate state dependency. Instead of relying on predefined taint sources, *InsFinder* employs predefined taint sinks and uses a backward propagation approach to identify external states, including states of other DApps and external transaction parameters. Then, *InsFinder* collects all check bytecode related to them (Section IV-A). To identify ISD-related security checks, *InsFinder* restores the source code related to the check bytecode. Then, *InsFinder* uses the LLM to understand the purpose of source code. Besides, *InsFinder* employs a chain-of-thought (CoT) mechanism [17] to enhance the stability and reasoning ability of the LLM (Section IV-B). To verify the effectiveness of checks, rather than directly evaluating the validity of checks, *InsFinder* uses fine-grained access control and static analysis to find the external state entry points, like publicly accessible functions, to verify checks (Section IV-C).

To validate the effectiveness of *InsFinder*, we build a dataset of 93 ISD incidents reported in DeFiHackLab dataset [18]. To the best of our knowledge, this dataset encompasses all ISDs that have occurred up to April 30, 2025. Experiment results indicate that *InsFinder* achieved a precision of 84.7% and a recall of 77.4% on this dataset, outperforming advanced tools such as GPTScan [14] and SmartReco [12]. Additionally, we collect 3,638 contracts from 122 DApps in Immunefi [19] for in-the-wild ISD analysis. The results show that *InsFinder* successfully detects 165 highly suspicious ISDs, affecting more than 2 million USD.

To promote DApp security development, we release the artifact of *InsFinder* and the dataset[1]. In summary, this paper makes the following contributions:

- We propose *InsFinder*, a new framework for detecting Insecure State Dependency (ISD) in DApps.
- We design a new backward taint analysis mechanism for dynamic cross-DApp analysis. This mechanism allows us to achieve precise multi-source state dependency tracking.
- We develop a new approach to complement the semantics of smart contract bytecode from source code. With its help, *InsFinder* can extract more fine-grained information for bytecode-level analysis (e.g., detecting ISDs).
- We perform extensive evaluation to verify *InsFinder*. The results indicate that *InsFinder* can detect ISDs while keeping low false positives and false negatives.

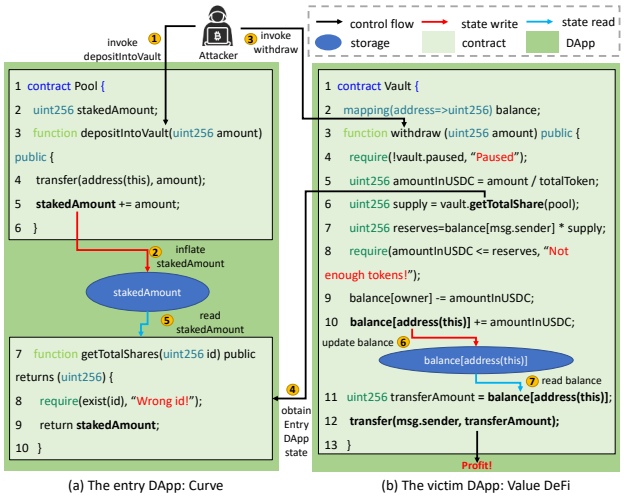
II. Background and Motivation

A. Background

Smart contracts and DApps. A smart contract is a piece of code, e.g., writing in Solidity [20] and Vyper [21], that can automatically run on the blockchain like Ethereum [22]. DApps use blockchain to store data and smart contracts for data processing. There are two types of roles in blockchain: user accounts (or Externally Owned Accounts, EOAs) and contract accounts. Due to the anonymity of blockchain, each account is assigned a unique identifier called an address for differentiation. To trigger functions of smart contracts, it is first necessary to compile the smart contract into bytecode and deploy the bytecode on the blockchain. Subsequently, when EOAs and contract accounts want to invoke this smart contract, they can send external and internal transactions respectively. Then, the blockchain will execute the bytecode based on the transaction input and blockchain state.

Insecure State Dependency. ISD is a new type of security threat in DApp interactions, which is caused by missing effective checks on external states in smart contracts. Due to the diverse functionalities of DApps, the attack surface of ISD is extremely large. In the meantime, the security implications of ISD may vary, such as price manipulation, business logic flaws, and read-only reentrancy. Besides, as a cross-DApp vulnerability, the triggering paths of ISD involve multiple DApps and are more obscure. As shown in Fig. 1, in particular, the victim function does not invoke the entry function during its execution, and vice versa. Therefore, compared with vulnerabilities occurred in contract(s) of a single DApp, like state-reverting [23], the detection of ISD is more difficult.

[1]https://github.com/zzz-sysu/InsFinder



(a) The entry DApp: Curve (b) The victim DApp: Value DeFi

Fig. 2: Motivating example with two critical code snippets related to ISD, along with key data flows and control flows.

Currently, due to the lack of effective detection methods, ISD occurs frequently, resulting in 93 recorded attacks and over 340 million USD in losses.

Transfer-related checks. Since anyone can call smart contracts, these contracts typically incorporate security checks to ensure the correct execution of the program [24]. Among them, transfer-related check (as we call T-Check) is one of the most important verifications for ensuring the execution of smart contracts. In detail, T-Check primarily examines variables associated with transfers, such as verifying whether the user has sufficient funds to transfer. To implement T-Checks, from the perspective of the source code, taking Solidity as an example, developers can use three keywords: `require`, `if`, and `assert`. During compilation, the compiler will convert the T-Check into a bytecode sequence that contains at least one check bytecode, such as `ISZERO`, `GT`, and `LT`. It is important to note that during the compilation process, the compiler may add additional checks for statements. For instance, in the case of a subtraction, the compiler may include a check to avoid underflow. Moreover, T-Checks are crucial for detecting ISD. We will show more details in Section II-B.

B. Motivating Example

Fig. 2 is an example of ISD adapted from a real-world security incident in Value DeFi [25], which results in a loss of over 11.4 million USD. The example involves code snippets for two key DApps, the victim DApp Value DeFi, and entry DApp Curve. The victim DApp is attacked due to ISD, while the entry DApp includes the critical states that the victim DApp relies on. Besides, the entry DApp is the entry point of ISD. Note that for easy understanding, we only select the most critical parts related to ISD for clarity, and the actual attack process is more complex. Additionally, we convert Curve's Vyper contracts into Solidity for easier readability.

In detail, the victim DApp's function `withdraw` relies on the state `stakedAmount` of the entry DApp to calculate the user's total asset value (line 6-7 in Fig. 2(b)). Then, the victim DApp uses this value to implement a T-Check (line 8 in Fig. 2(b)) to determine whether to transfer funds to the user. However, since this check includes multiple external states, it can be bypassed. More specifically, in the contract `Pool` of the entry DApp, the function `depositIntoVault` allows users to deposit funds into the DApp, which updates `stakedAmount` accordingly. To exploit ISD, the attacker first calls function `depositIntoVault` (step 1) to inflate `stakedAmount` (step 2). The attacker then calls function `withdraw` to take funds (step 3). Since the attacker has manipulated `stakedAmount`, the value of `reserves` (line 7 of Fig. 2(b)) will be inflated (step 4-5). Finally, the attacker bypasses the T-Check and profits through transfers and withdrawals from `Pool` (step 6-7).

Key observation for ISD detection. Through the analysis of all ISDs that have occurred so far, we summarize the essence of ISD detection lies in two aspects: (1) finding necessary transfer-related checks (T-Checks), and (2) validating the effectiveness of the implemented T-Checks.

- *Existence of T-Checks.* Attackers can always obtain substantial profits through token transfer after exploiting ISD. Therefore, implementing T-Checks is mandatory for sterilizing external states related to trading, and preventing potential ISD.
- *Validity of T-Checks.* All ISDs are caused by the failure or absence of T-Checks. The T-Check is valid only if there is at most one external state in the check. Specifically, when multiple external states are present in a T-Check, an attacker can attempt to manipulate these states to influence the result of the check. For example, the check in line 8 of Fig. 2(b) is intended to ensure that the amount transferred to users is within a reasonable range. However, since the check involves two external states, `stakedAmount` and `amount`, attackers can control `amount` through function parameter and call `depositIntoVault` to manipulate `stakedAmount`. Finally, they can bypass this check to profit.

III. Design of *InsFinder*

A. Design Choices of *InsFinder*

Finding check bytecode of critical external states. DApps heavily rely on external states from other DApps, but not all external states are necessarily related to ISD (e.g., state `paused` in line 4 of Fig. 2(b)). To accurately collect all check bytecode of ISD-related external states, *InsFinder* uses a dynamic multi-source tracer which combines dynamic cross-DApp taint analysis and transfer-based backward propagation method. In detail, *InsFinder* sets the transfer amount as a taint sink and employs backpropagation to find associated external states. The observation here is that while there are many origins of ISD, all ISD attacks directly result in economic losses through trading. Thus, the endpoints of ISD

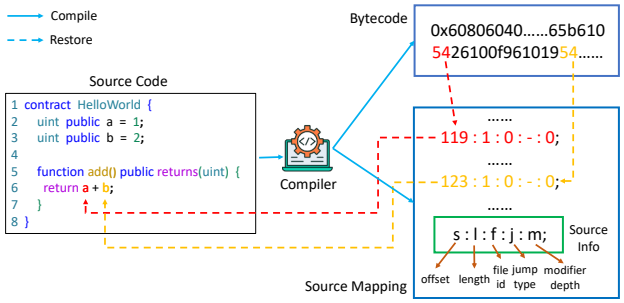


Fig. 3: The workflow of smart contract compilation and source mappings. The source code generates bytecode and source mappings through the compiler. The bytecode can be restored to its corresponding source code through source mapping.

are relatively fixed, and *InsFinder* can identify taint sources associated with ISD accurately. Besides, since ISD is a cross-DApp security threat, it is feasible to consider inter-DApp data dependencies as secure. Therefore, we regard states of other DApps and external transaction arguments as the external states of ISD. Additionally, compared to static taint analysis, dynamic taint analysis can perform more precise data tracing by incorporating execution context information. We will give more details about this process in Section IV-A.

Identifying T-Checks. As the EVM operates based on bytecode, *InsFinder* can only obtain the bytecode related to the checks through the multi-source tracer. To identify T-Checks through check bytecode, *InsFinder* employs a semantic enriched T-Check analyzer. In detail, *InsFinder* utilizes source mapping to enhance the semantics of check bytecode and employs LLMs to identify T-Checks. As shown in Fig. 3, during the compilation, the compiler compiles the source code into bytecode and source mapping. The source mapping records the mapping information from bytecode to source code [26]. Specifically, each line of source mapping indicates the position of a bytecode in the source file. Besides, each line contains useful information: s represents the offset of the starting position in the source file, l indicates the range of the source code, f corresponds to the source file index, j denotes the type of jump instruction, and m represents the current modifier depth. For example, the red part in Fig. 3 indicates that the source code corresponding to the bytecode `0x54` is a string of length 1, starting at an offset of 119 bytes in source file 0, which is a. Besides, this bytecode is not related to any jump instruction and is not within any modifier. Therefore, source mapping can accurately restore bytecode back to the source code, improving the semantics of the bytecode.

Then, *InsFinder* employs LLMs to understand the corresponding source code semantics, such as determining whether the source code represents a check. To enhance the stability and reasoning capability of LLMs, *InsFinder* employs a chain-of-thought (CoT) mechanism [17]. CoT facilitates LLMs by breaking down complex problems into multiple sub-problems, allowing the LLMs to think through the issue step by step [27]. Finally, *InsFinder* can make a comprehensive assessment of the purpose of the checks and accurately identify all T-Checks. More details are shown in Section IV-B

Determining the existence of ISD. As mentioned before, obtaining specific execution context information is by no means trivial. To verify the effectiveness of T-Checks, *InsFinder* introduces an ISD entry point checker. In detail, the checker utilizes dynamic execution information to reconstruct the complete call chain. Then, it transforms the verification of T-Checks into the detection of whether external state entry points exist. The insight here is that the triggering of ISD requires manipulable external states. Therefore, if there are entry points, such as publicly accessible functions for modifying the external states in the check, it becomes feasible to exploit the ISD. More details are illustrated in Section IV-C

B. Workflow of InsFinder

Fig. 4 shows the workflow of *InsFinder*. *InsFinder* receives the DApp contract addresses as input, and outputs the ISD detection results. Specifically, *InsFinder* first fetches transactions and uses a dynamic multi-source tracer to monitor the execution of EVM. When the EVM executes transactions, the tracer analyzes and records each bytecode and the corresponding data. After the transaction execution, the tracer restores the data dependency and reports all check bytecode that needs to be tested. Then, the semantic enriched T-Check analyzer recovers the source code information from bytecode. After that, the analyzer uses LLM to filter all T-Checks. Then, the ISD entry point checker determines how many entry points are in these checks. In detail, if there is more than one manipulable external state in the check, *InsFinder* outputs the involved external states along with the corresponding entry points.

Again, taking Fig. 2 as an example, *InsFinder* first employs the dynamic multi-source tracer to collect and replay transactions of DApp Value on the chain. When executing transaction of `withdraw`, tracer identifies all external states that the transfer amount (e.g., `transferAmount`) is dependent on, e.g, `amount` and corresponding inherited states `amountInUSDC`. *InsFinder* then gathers the check bytecode that uses these states. In the semantic enriched T-Check analyzer, *InsFinder* recovers source code and finds that the check in line 8 of Fig. 2(b) contains `amountInUSDC` and is a T-Check. In the ISD entry point checker, *InsFinder* determines that this check can be potentially bypassed due to the involvement of another external state `stakedAmount`. Then, the checker analyzes these external states individually and verifies if entry points exist. Since `depositIntoVault` can modify `stakedAmount` and `amountInUSDC` can be manipulated by attackers, *InsFinder* reports this ISD.

IV. Approach Details

A. Dynamic Multi-Source Tracer

In this module, *InsFinder* employs three data managers, called $MSource$, $MStorage$, and $MCall$, to track the execution process and uses dynamic cross-DApp taint analysis to find all external states. More specifically, data managers

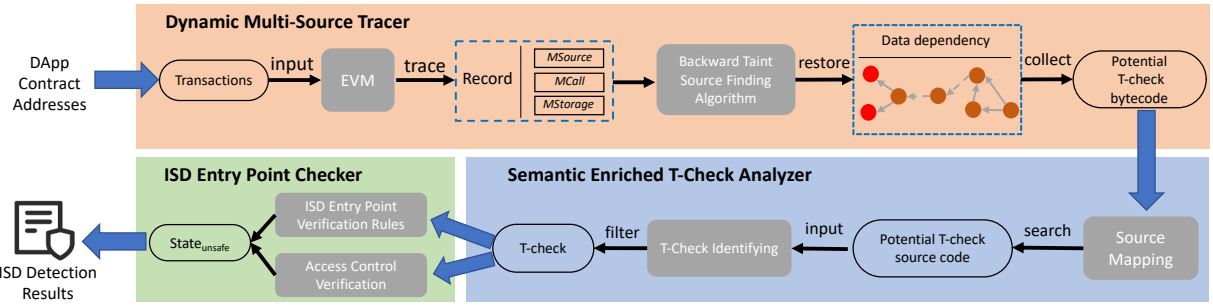


Fig. 4: The workflow of *InsFinder*.

mainly track data in the stack and memory of the EVM, and storage in blockchain, as these are the most common data sources and most closely associated with ISD. Besides, it is worth noting that backward propagation for identifying DApp data dependencies is by no means trivial. As shown in Fig. 5, during DApp interactions, external states can be passed cross-variable, cross-contract, and even cross-storage. Moreover, the EVM and storage do not contain data sources. In the end, data tracking may be lost during analysis. Once data tracking is broken, *InsFinder* may miss external states. Note that, *InsFinder* references SmartReco [12] to perform cross-DApp analysis based on the DApp builder. Since this is not the contribution of this paper, we do not elaborate in this section. We will discuss the influence of DApp builders in Section V-B.

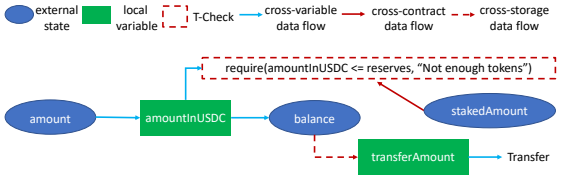
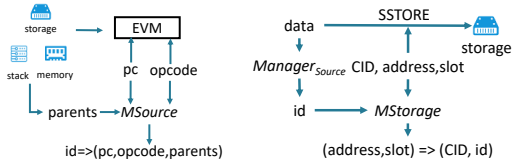


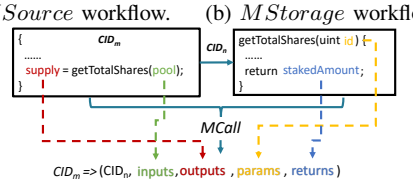
Fig. 5: The data flow of key external states related to ISD in `withdraw` of Fig. 2. `stakedAmount` in entry DApp is passed through cross-contract data flow. `amount` transmit to `transferAmount` via cross-storage data flow.

Inter transaction source tracing. Once the transaction begins execution, *InsFinder* monitors and processes the data based on the purpose of bytecode. Specifically, since the EVM generates new memory and stack for each transaction (both internal and external transactions), *InsFinder* creates $MSource_{CID}$ to manage data for each call CID. As shown in Fig. 6(a), $MSource_{CID}$ is a mapping: $id => (pc, opcode, parents)$. id is the identifier for this data. The pc denotes the current program counter, $opcode$ indicates the instruction related to the generation of id, and $parents$ is a list from which id originates. Taking the code in line 6 of Fig. 3 as an example, suppose the id of `a`, `b` and `c` is id_a, id_b and id_c respectively, the pc and $opcode$ related to the calculation are pc_{ADD} and `ADD`, the CID is n. *InsFinder* will generate $id_c => (pc_{ADD}$, `ADD`, $[id_a, id_b])$ to $MSource_n$. Finally, *InsFinder* recursively searches for the $parents$ of id in $MSource_{CID}$ to perform cross-variable analysis and find the sources.

Intra transaction source tracing. When data is passed across transactions, including cross-storage and cross-contract, data dependencies are interrupted. Specifically, once a state in storage is modified in one transaction, subsequent transactions can directly read the updated value from storage. To restore cross-storage data dependency, *InsFinder* uses $MStorage$: $(address,slot) => (CID, id)$. Here, $address$ indicates the address of the updated storage, $slot$ specifies the location where the updated state is stored, CID is the call for executing operations, and id indicates the data to be stored. As shown in Fig. 6(b), $MStorage$ records all storage write operations, such as `SSTORE`. When *InsFinder* wants to verify whether the data in the $slot$ of $address$ has been updated, it queries $MStorage$ to check for related records.



(a) $MSource$ workflow. (b) $MStorage$ workflow.



(c) $MCall$ workflow.

Fig. 6: Workflow of three data managers used by *InsFinder* during the multi-source tracing.

As shown in Fig. 6(c), *InsFinder* uses $MCall$ to achieve cross-contract data dependency recovery: $CID =>(CID_{new}$, $inputs$, $outputs$, $params$, $returns)$. Specifically, when the call CID performs the call bytecode during the execution, such as `CALL`, *InsFinder* assigns CID_{new} to the new call. Simultaneously, *InsFinder* records the input id list $inputs$ and the output id list $outputs$ of CID. When executing CID_{new}, *InsFinder* records the parameter id list $params$ and

the return id list $returns$ of CID_{new}. For example, when `withdraw` calls `getTotalShares` in Fig. 2, the CID of them are CID_1 and CID_2 respectively, *InsFinder* records this call as CID_1 =>(CID_2, [id_{pool}], [id_{supply}], [id_{id}], [$id_{stakedAmount}$]). Whenever *InsFinder* wants to determine whether data in CID is related to a cross-contract call, it checks whether the data has a corresponding record in $MCall$.

Finding critical check bytecode . *InsFinder* uses the following rule to locate taint sinks:

$$Transfer(r, a)\&\&r = tx.origin => Sink(a) \quad (1)$$

where $Transfer$ represents the transfer operation, r denotes the recipient, a indicates the transfer amount, $tx.origin$ is the sender of the external transaction, and $Sink$ marks the taint sink. This rule indicates that when *InsFinder* detects a transfer sent to the sender of an external transaction, the transfer amount is related to ISD.

Algorithm 1 Backward Taint Source Finding Algorithm.

Input: Transfer amount id id_{amount}, transaction's DApp D_{name}
Output: Taint source list $L_{external}$

```
cache := [id_amount]
while isNotEmpty(cache) do
    id := cache.pop( )
    parents := MSource.analysis(id)
    cross_contract := MCall.analysis(id)
    cross_storage := MStorage.analysis(id)
    if isEmpty(parents) && isEmpty(cross_contract)
    && isEmpty(cross_storage) then
        L_external.pushIfNotSameDApp(id, D_name)
    else
        cache.extendWith([cross_contract,
        cross_storage, parents])
    end if
end while
```

When a taint sink is found, *InsFinder* employs a backward propagation method to identify the taint sources, and the detailed process is shown in Algorithm 1. Specifically, $cache$ is the list of id to be analyzed, initially including only taint sinks. *InsFinder* continuously analyzes id in $cache$. If id has no parents, cannot be linked to other data through cross-variable, cross-contract, and cross-storage analysis, and the DApp corresponding to the id is different from the DApp of the external transaction, then id is considered as a taint source, and *InsFinder* adds id to $L_{external}$. Otherwise, *InsFinder* pushes them into $cache$ for further analysis. Besides, only searching for checks based on $L_{external}$ may lead to false negatives, as DApps can transform states into other variables for verification, like changing `amount` to `amountInUSDC`. Therefore, *InsFinder* considers all states in $L_{external}$ as well as their child states as external states under test. Then, *InsFinder* collects all check bytecode related to these states.

B. Semantic Enriched T-Check Analyzer

Directly analyzing bytecode makes it difficult to understand the intent of the checks. Therefore, *InsFinder* employs source mapping and LLM for bytecode analysis.

Converting bytecode to source code. The source code corresponding to the bytecode is usually just a variable name or statement. For example, the red 0x54 in Fig. 3 can only yield the string `a` through source mapping, without any additional information. However, without complete function source code, it is still challenging for LLM to determine whether a check statement is related to transfers. Therefore, *InsFinder* simultaneously restores both the statement and the function source code corresponding to the bytecode. More specifically, to find the source code corresponding to the check bytecode, *InsFinder* first compiles the contract into the bytecode, and generates a source mapping file. *InsFinder* then searches the source mapping file for the location of the check bytecode and corresponding function. Subsequently, *InsFinder* retrieves the source code of them as strings through a slicing method.

T-Check Finding Prompt Template

System: You are a smart contract security auditor. Your task is to evaluate whether the code meets the requirements of each step and output the result for each step. You must strictly follow the user-defined rules. Please write your reasoning results in [Answer].

— — — — — — — — — — — —

CoT of T-Check finding: [Function] provides the complete function code, and [Check] is (part of) a statement that needs to be checked. You need to think step by step to determine whether [Check] satisfies the following requirements:
Step 1. Find the statement corresponding to [Check] in [Function]. If the statement does not have keywords for checking, such as `if`, `require`, or `assert`, add `Requirement_NoCheck` to the [Answer]. Otherwise, add `Requirement_Check` to the [Answer].
Step 2. Use the code in [Check] to determine whether [Check] is a security check or execution control. The branches after security checks typically include exception handling or return whether the data is abnormal, while the branches following execution control include state read and write, calculation, and function call. If [Check] is not a security check, add `Requirement_Execute` to [Answer]. Otherwise, add `Requirement_Security` to [Answer].
Step 3. Determine whether [Check] is used for transfer decisions. Typically, checks related to transfers will involve token amounts or prices, which you can infer from variable names and function logic. If you believe this is a transfer-related check, add `Requirement_Transfer` to [Answer]. Otherwise, add `Requirement_Logic` to [Answer].

— — — — — — — — — — — —

[Function]
[Check]

Fig. 7: Prompt template for finding T-Checks.

Identifying T-Check. Determining whether a check is transfer-related requires substantial expert knowledge. To prevent the large language model (LLM) from experiencing

hallucinations and to enhance its reasoning capabilities in this context, *InsFinder* combines expert knowledge with a chain-of-thought (CoT) mechanism to design prompts.

Fig. 7 illustrates the detailed prompt template. Due to the inherent uncertainty of the LLM [14], *InsFinder* instructs the LLM to output the result of each step into [Answer]. Then, *InsFinder* makes a comprehensive assessment of whether the statement is a T-Check. Specifically, *InsFinder* adds the functions and statements to be checked into [Function] and [Check], respectively. To eliminate the additional checks introduced by the compiler, if [Check] does not contain check keywords, such as `if` and `require`, *InsFinder* instructs the LLM to add `Requirement_NoCheck` to [Answer]. Otherwise, LLM adds `Requirement_Check`. Next, some checks are intended to control branch execution, such as determining to execute statements after `if` or `else`, and are unrelated to the sanitization of external states. Thus, *InsFinder* instructs the LLM to determine the purpose of [Check] based on branch intention. If LLM believes [Check] is intended to control the branch execution, it will add `Requirement_Execute` to [Answer]. Or, it adds `Requirement_Security`. Consequently, *InsFinder* asks the LLM to understand the semantics of [Check] based on [Function] and return the purpose of the check. If the LLM considers [Check] as a T-Check, it will add `Requirement_Transfer` to [Answer]. Otherwise, it adds `Requirement_Logic`. Finally, only when [Answer] contains `Request_Check`, `Request_Security`, and `Request_Transfer` simultaneously, *InsFinder* considers the check as a T-Check.

Access Control Prompt Template

System: (Same as in Fig. 7, omitted for brevity.)

Prompt: You need to determine whether [Function] has access control strategies, like only being accessed by the owner. If so, add `Requirement_True` to [Answer]; otherwise, add `Requirement_False` to [Answer]. Note that, it is considered invalid if the access control allows ordinary users to access it.

[Function]

Fig. 8: Prompt template for verifying access control.

C. ISD Entry Point Checker

For each T-Check, *InsFinder* analyzes and collects all the external states involved. When identifying more than one external state in a T-Check, *InsFinder* will further analyze all these external states and use a set called $State_{unsafe}$ to collect all manipulable external states. Specifically, there are two types of external states related to ISD. One is passed through the external transaction, like function parameters, and the other is obtained by reading other DApps' states. For external states acquired through the external transaction, *InsFinder* directly considers them unsafe and adds them to $State_{unsafe}$, as these states can be arbitrarily specified by an attacker. For external states obtained by reading other DApps' states, attackers must identify publicly available functions that can modify this state. Thus, *InsFinder* applies the following rule:

$$\begin{aligned} \exists func \in Func_{write}(state) \quad \&\& \quad \neg Access(func) \\ => state \in State_{unsafe} \end{aligned} \tag{2}$$

where $state$ is the external state to be analyzed, $Func_{write}$ is the set of functions that can modify $state$, $func$ is the function in $Func_{write}$ to be analyzed, $Access$ represents the access control rules in $func$. This rule indicates that when a function is found that can modify external states and has no access control strategy, *InsFinder* considers the function as the entry point of the state and marks the state as exploitable.

More specifically, to find entry points for DApp states, firstly, *InsFinder* uses state read-write dependencies to search the contract to collect all functions capable of modifying that state. These functions are considered potential ISD entry point functions. Then, *InsFinder* conducts access control analysis on these functions to eliminate potential false positives. For example, a function that can only be called by the owner can be considered safe. Due to the complex and flexible logic of DApps, there are various ways to implement access control, such as using `modifier`, `if`, or `require` statements. As a result, traditional pattern-based methods [10], [28] tend to have low accuracy. Thus, *InsFinder* leverages the LLM to determine whether a function has access control. Fig. 8 presents the specific prompt template. Since some access controls may still allow ordinary users to access functions, *InsFinder* considers access controls that permit only owners or administrators to invoke as effective.

$$Count(State_{unsafe}) > 1 => ISD\ exists \tag{3}$$

After the above analysis, when there is more than one state in $State_{unsafe}$, *InsFinder* concludes that the check can be bypassed and ISD exists. Then, *InsFinder* reports the check statement, the related external states, and the entry points for those external states.

V. Evaluation

In this section, we first present the dataset used in the evaluation and introduce our evaluation setup. Then we show the evaluation results of *InsFinder*.

A. Implementation

Dataset. We use following datasets to perform our evaluation.

- **Manual-labeled ISD Dataset** ($D_{labeled}$). This dataset contains 93 reported ISD attacks. We mainly select price manipulation, business logic flaws, and read-only reentrancy from the DeFiHackLab dataset [18], as they are the most common and harmful vulnerabilities in DApp interactions. Besides, the DeFiHackLab dataset includes the most comprehensive and significant smart contract attack events currently available and provides verifiable Proof-of-Concept (PoC) scripts. We invite three experts with at least two years of experience in smart contract auditing to annotate the dataset. Specifically,

we first conduct an initial screening through related keyword matching of terms such as price, logic, and reentrancy. Then, experts confirm the root cause of each vulnerability based on PoC, retaining only the attack caused by cross-DApp state dependencies. Besides, we let experts point out the T-Check in these DApps. Additionally, we only classify an attack as an ISD if all three experts agree. Finally, we obtain 93 ISD victim contracts in $D_{labeled}$ and 118 T-Checks in these attacks. To the best of our knowledge, this dataset covers all ISDs that have occurred up to April 30, 2025. The distribution of vulnerabilities is shown in Table I.

- **Popular DApp Dataset ($D_{unknown}$).** To further analyze *InsFinder*'s ability to detect unknown ISDs, we collect 3,638 contracts from 122 projects on a well-known bug bounty platform, Immunefi [19], for testing. These contracts and projects are of high quality and are representative.

Implementation. We implement *InsFinder* with approximately 1,300 lines of Python and 4,000 lines of Rust. Specifically, the dynamic multi-source tracer in *InsFinder* is based on the transaction replay module of SmartReco [12]. *InsFinder* uses smart contract compilers, currently supporting Solidity and Vyper, to extract source mapping and state dependencies, and uses OpenAI's GPT-4o model [29] to filter T-Checks and identify access control. Parameters in GPT-4o are set to default values. *InsFinder* clears context before each LLM request.

Evaluation metrics. We use TP (True Positive) to represent correctly identified contracts ISDs. We use FP (False Positive) to signify the incorrect labeling of a contract without vulnerability as ISD. We use FN (False Negative) to denote misclassifying a given contract that contains ISD as harmless.

Research questions. We primarily focus on the following research questions.

RQ1. How effective is *InsFinder* in detecting ISD?
RQ2. What is the impact of each module on detecting ISD?
RQ3. What is the overhead of *InsFinder* for detecting ISD?
RQ4. Can *InsFinder* effectively detect real-world ISD?

B. Effectiveness of InsFinder

To answer RQ1, we evaluate *InsFinder* based on the $D_{labeled}$. In detail, we let three experts manually confirm the test results of *InsFinder*. We consider it a true positive only when *InsFinder* accurately reports the specific location of the attack, e.g., entry points of the external states. Since there may be more than one entry point for exploiting ISD, as long as the path reported by *InsFinder* is potentially feasible, we consider it a true positive. Table I shows the final results. From the table, we can find that *InsFinder* successfully detects 72 ISDs and has a precision of 84.7% and a recall of 77.4%. The LLM's analysis errors lead to 6 false positives and 10 false negatives, and we will comprehensively analyze in Section V-C.

False positives. Through the analysis of false positives, we identify the main reason as over-analysis caused by incomplete DApp information. *InsFinder* follows SmartReco to manually collect DApp contracts and extract the builders to determine contract identification and perform cross-DApp analysis. However, manual collection may miss some builders of DApps, leading to the failure of contract identification for the same DApp. This can lead *InsFinder* to consider the states of the same DApp as unsafe, resulting in a more stringent analysis. Note, the anonymity of blockchain means that the contract itself does not contain DApp information. Besides, identifying DApp boundaries through builders is currently the most effective cross-DApp analysis method [12].

TABLE I: The price manipulation (PM), business logic flaws (BLF), and read-only reentrancy (ROR) detection results of *InsFinder*, GPTScan, SmartReco based on $D_{labeled}$.

		InsFinder			GPTScan			SmartReco		
		TP	FP	FN	TP	FP	FN	TP	FP	FN
PM	40	32	5	8	6	1	34	-	-	-
BLF	45	34	8	11	6	2	39	-	-	-
ROR	8	6	0	2	-	-	-	2	0	6
Total	93	72	13	21	12	3	73	2	0	6

False negatives. The reasons for false negatives are as follows. (1) Tool analysis failure. Due to a lack of specific compilation parameters, some contracts fail to compile locally, resulting in the absence of source mapping and state dependency. (2) Lack of source code. *InsFinder* relies on the source code to generate source mapping files. If the contract does not disclose its source code, *InsFinder* cannot process it. However, with complete code, the developers can avoid this issue.

Comparison with advanced tools. In addition, to better demonstrate the effectiveness of *InsFinder*, we compare *InsFinder* with GPTScan [14] and SmartReco [12]. GPTScan is currently the SOTA tool in using LLM to detect business logic flaws and price manipulation. SmartReco is the most effective tool for detecting read-only reentrancy vulnerabilities. Notably, we do not select other static analysis tools such as Slither [15] or Sailfish [30] because they lack the capability for semantic analysis. Additionally, we exclude dynamic analysis tools like ityFuzz [31] and Confuzzius [11], as they cannot perform multi-function fuzzing, not mention to detect ISD. Specifically, we obtain the latest versions of these two tools and use their default configurations. In addition, to eliminate the influence of different LLMs on the experiment, we change GPTScan's default LLM from GPT-3.5 to GPT-4o, which is more effective. Besides, we let three experts evaluate the results reported by the two tools using the same standards as *InsFinder*.

As shown in Table I, GPTScan exhibits higher false negatives. Through in-depth analysis, we discover that GPTScan, as a static analysis tool, lacks precise execution context and DApp boundaries. As a result, LLM lacks sufficient information for judgment, ultimately affecting the decision of GPTScan. SmartReco effectively reduces false positives by identifying DApp boundaries. However, SmartReco has many false negatives, as it relies on multi-function transactions for detection and support limited to Solidity.

C. Effectiveness of each module in InsFinder

Stability of LLM analysis. We collect 118 T-Checks and randomly select 32 other checks and 100 functions from D_{label}

to test LLM stability in recognizing T-Checks and access control. We compare the results of testing a single round versus independently testing 3 rounds and selecting the most common value under GPT-4o and DeepSeek R1 [32]. As shown in Table II, through carefully designed prompts and CoT, the difference between single-round testing and multi-round testing, as well as between different models, are minimal. Through the analysis, we find that the main reason for identification errors is due to the insufficient source code. In detail, security checks or access control analyses may involve function calls to make decisions. However, *InsFinder* does not provide the implementation of these functions. As a result, the LLM infers their functionality based on the function names, which may negatively impact the final judgment. The design choices are as follows: On the one hand, prior studies [33], [34], [35] have demonstrated that excessively long inputs are more likely to induce hallucinations in LLM. On the other hand, this approach can effectively reduce token costs and improve the usability of *InsFinder*.

TABLE II: T-Check and access control recognition results vary across different LLM models and test rounds.

		T-Check			Access Control		
		TP+TN	FP	FN	TP+TN	FP	FN
GPT-4o	1 Round	137	2	11	91	5	4
	3 Rounds	140	1	9	96	3	1
DeepSeek	1 Round	134	3	13	92	5	3
	3 Rounds	137	3	10	95	3	2

Effectiveness of CoT analysis. To evaluate the effectiveness of CoT in identifying T-Checks, we replace it with a standard prompt (without CoT). Specifically, we provide the function and the corresponding check as input to the LLM, and ask it to output Yes if it believes the check qualifies as a T-Check, and No otherwise. We conduct experiments on 150 collected checks. without CoT successfully identified only 34 T-Checks, with 7 false positives and 84 false negatives. Since identifying T-Checks is a challenging task, the absence of CoT reasoning significantly hindered the LLM's ability to reason effectively, resulting in poor performance.

Effectiveness of dynamic multi-source tracer. To validate the dynamic multi-source tracer in identifying taint sources, we remove it and refer to the taint definition from DeFiTainter (without MST). In detail, we set function `balanceOf` as the taint source and `transfer` as the taint sink. If there is a data flow from `balanceOf` to `transfer`, we proceed with the data in `balanceOf` for further analysis. As shown in Table III, the predefined taint source causes without MST to overlook potential external data, leading to a significant number of false negatives. However, it is worth noting that irrelevant taint information does not introduce a lot of false positives. After an in-depth analysis, we find that on the one hand, the T-Check analyzer helps without MST understand the intent of `balanceOf`. On the other hand, dynamic analysis enables more precise data tracking.

Effectiveness of semantic enriched T-Check analyzer. To verify whether the semantic enriched T-Check analyzer can understand check bytecode, we remove it and conduct T-check verification solely based on the data sources involved in checks (without SEA). Specifically, if a T-check uses contract own state to verify external states, we consider this check valid. The results are shown in Table III. Pattern-based methods do not identify the purpose of checks based on the semantics. As a result, the false negative rate arises.

TABLE III: Detecting results of *InsFinder*, without dynamic multi-source tracer (without MST) and without semantic enriched T-check analyzer (without SEA) based on $D_{labeled}$.

	TP	FP	FN	Precision	Recall
without MST	17	12	76	58.6%	18.3%
without SEA	0	0	93	-	-
InsFinder	72	13	21	84.7%	77.4%

D. Overhead of InsFinder

To answer RQ3, we evaluate the efficiency of *InsFinder* and compare the financial costs of *InsFinder* and GPTScan.

Efficiency. *InsFinder* takes an average of 259.7 seconds to find ISD. Since *InsFinder* is an online analysis tool, it fetches data and contract information from the blockchain. Besides, *InsFinder* needs to query the LLM online for results. However, *InsFinder* employs a caching mechanism to reduce network overhead. For instance, downloading and compiling contract code is a one-time cost. When the information for the contract is needed again, *InsFinder* directly retrieves the cached results. Additionally, the tracer in *InsFinder* processes data solely based on the execution results of the interpreter, without adding any overhead to the EVM execution. Therefore, the time overhead of *InsFinder* is considered acceptable.

Financial costs. On average, *InsFinder* uses a total of 1.85 USD, and each DApp costs on average 0.024 USD. In addition, GPTScan spends a total of 3.48 USD, and for each DApp, it uses an average of 0.074 USD. This indicates that *InsFinder* spends fewer tokens, and the cost is acceptable. We identify the reason as follows: combining with multi-source tracer and source mapping, *InsFinder* can provide precise code snippets related to T-Checks and access control, and it only needs the LLM to return the final answer of each step. As a result, *InsFinder* requires only a minimal number of tokens.

E. Large-scale Analysis for ISD in the Wild

To answer RQ4, we use $D_{unknown}$ to evaluate *InsFinder*. Finally, *InsFinder* detects 203 ISD. We let three experts independently evaluate the results reported by *InsFinder*. If they determine that a case is an exploitable ISD, they are required to construct a PoC to demonstrate it. Ultimately, 165 ISDs are confirmed by the experts, with a precision of 81.3%. After our analysis, the total value affected by these ISDs exceeds 2.6M USD. Due to time constraints, at the time of writing, we have reported eight vulnerabilities to developers and have not yet received a response. Since confirming all ISDs in $D_{unknown}$ is difficult, we do not conduct recall evaluation. However,

```
1 contract Vault {
2   function borrow(uint256 amount, uint256 price) {
3     ...
4     uint256 onChainPrice = pair.getAmountOut() /
          pair.totalSupply();
5     uint256 diff = onChainPrice / price;
6     require (1 - diff < threshold, "PRICE_GAP");
7     transfer(msg.sender, price * amount);
8   }
9 }
```

(a) Smart contract with business logic flaws.

```
1 @external
2 @nonreentrant('lock')
3 def remove_liquidity(_amount, receiver):
4     ...
5     self.balance = self.balance - _amount
6     raw_call(receiver, _amount)
7     self.D = self.D - _amount / total_supply
8     ...
9 @external
10 def get_reserves():
11     return self.D, self.balance
```

(b) Smart contract with read-only reentrancy.

Fig. 9: Simplified real-world smart contracts with ISD.

the false negative evaluation in Section V-B demonstrates that *InsFinder* can cover most ISDs.

Case study 1. The contract in Fig. 9(a) contains business logic flaws. In detail, function `borrow` calculates the number of tokens to borrow to the user based on the arguments `amount` and `price`. To ensure that the user's expected price is within a reasonable range, `borrow` performs a T-Check (line 6 of Fig. 9(a)). However, since this check relies on multiple external states, such as `price` and `onChainPrice`, it can be bypassed. LLM can recognize this is a T-Check. However, since lacking knowledge of the execution context and data source, LLM arrives at incorrect conclusions. For example, in our multiple independent queries, GPT-4o consistently misidentifies this as a valid T-check. *InsFinder* can leverage the fine-grained static analysis to detect T-Check precisely.

Case study 2. Fig.9(b) depicts a read-only reentrancy written in Vyper. Function *remove_liquidity* allows users to withdraw funds (line 6 in Fig.9(b)). However, as the value of self.D is updated after the transfer, attackers can exploit this by calling function *get_reserves* to obtain incorrect values and profit in another DApp. SmartReco misses this vulnerability due to its support only for Solidity. *InsFinder* supports the analysis of multiple smart contract languages.

VI. Discussion

A. Discussion

Security impact of detected ISDs. It is important to note that not all ISDs will directly lead to exploitation, as attackers always rationally allow their attacks to profit [36], [37]. For example, when a liquidity pool has high liquidity, attackers require more funds and tolerate a higher possibility of failure. Conversely, when liquidity is lower, attackers can exploit with less capital and lower risk. Therefore, the exploitation of ISDs needs to be assessed based on the actual circumstances. *InsFinder* can accurately identify all potential external variables and T-Checks that may be bypassed. This capability effectively assists developers and auditors in pinpointing critical locations and potential exploit paths, allowing for a more accurate determination of whether ISDs exist.

Detecting ISD with deployed contracts. *InsFinder* primarily uses on-chain data to detect ISDs, as DApp interactions are hardly reconstructed in off-chain environments [12]. Besides, contracts that have been audited multiple times can still be vulnerable to attacks after deployment [38], [39], as the on-chain environment is complex and dynamic. Therefore, detecting deployed contracts can still help DApps find vulnerabilities in advance and reduce losses.

B. Threats to Validity

Internal validity. If the contract's source code or transaction is missing, *InsFinder* may fail to determine whether a check is transfer-related, leading to false negatives. However, since users tend to invest in open-source DApps, most DApps are currently open-source [40]. Besides, utilizing source code is practical, as *InsFinder* is mainly designed for DApp owners/third-party auditors. Both have full access to the source code. Moreover, similar to existing methods [9], [24], *InsFinder* can perform ISD detection without relying on on-chain data. Specifically, if a comprehensive off-chain testing environment can be provided, e.g, through fuzzing, *InsFinder* can detect ISDs along the execution paths. Therefore, we consider this impact to be limited. Additionally, *InsFinder* depends on the LLM to assess the purpose of checks. To mitigate the influence of the LLM on *InsFinder*'s detection, we carefully design prompts and employ a COT mechanism that allows the LLM to think step by step, enhancing the stability of the results.

External validity. Currently, *InsFinder* only supports EVM-based blockchains. While *InsFinder* can analyze multiple smart contract languages, such as Solidity and Vyper, different blockchains may have distinct architectures. Nonetheless, *InsFinder* has good scalability because it is not directly integrated into the EVM. Adjustments can be made based on specific differences for non-EVM blockchains. Additionally, *InsFinder* currently identifies taint sinks through transfers to specific accounts. Although ISDs are all related to transfers, attackers may hide their attacks through other methods, such as dispersing tokens across multiple accounts. In future work, we will explore how to cover these more complex transaction behaviors to enhance the applicability of *InsFinder*.

VII. Related Work

Smart contract vulnerability detection. Currently, many studies utilize program analysis techniques to detect vulnerabilities in smart contracts. Fuzzing tools trigger vulnerabilities by constructing inputs that meet specific criteria. For

instance, ContractFuzzer [41] employs black-box fuzz testing for vulnerability detection. However, due to the inefficiency of black-box testing, Harvey [42] enhances fuzz testing efficiency through gray-box testing. Since off-chain fuzzing cannot simulate real interaction scenarios, tools like ityFuzz [31] and icyChecker [9] use on-chain data for more realistic testing. Besides, SmartReco [12] proposes a multi-function-based fuzzing approach to identify vulnerabilities. Static analysis tools search for vulnerabilities by analyzing the source code or bytecode of contracts. Oyente [43] detects vulnerabilities through symbolic execution to assess path reachability. AChecker [10] and Gigahorse [44] extract vulnerable bytecode sequences with heuristics and pattern matching.

Understanding code semantics of smart contract. Unlike traditional software vulnerabilities, smart contract vulnerabilities typically result in logic errors that allow for the extraction of greater profits rather than causing program crashes. Therefore, understanding the semantics of smart contracts is highly beneficial for vulnerability detection. However, most tools [8], [16] employ predefined rules. For example, when the execution of a smart contract aligns with these rules, a particular behavior is deemed to occur. Consequently, this approach often results in high false positive rates and has significant limitations. With the maturation of LLM, some studies [14], [45], [46] have begun using LLMs to detect vulnerabilities. However, current LLMs are primarily suited for vulnerability detection in source code [47] and are not effective for understanding the semantics of bytecode. To fill this gap, *InsFinder* utilizes source mapping to convert bytecode back into source code, thereby assisting LLMs in understanding the semantics of bytecode.

VIII. Conclusion

In this paper, we present *InsFinder*, a tool focused on detecting ISD in DApps. Specifically, we employ a dynamic cross-DApp taint analysis method to locate potential external states related checks bytecode. Consequently, *InsFinder* uses source mapping to enrich the semantics of bytecode and then uses the LLM to filter T-Checks. Ultimately, *InsFinder* detects ISDs by identifying entry points. Experimental results demonstrate that *InsFinder* outperforms state-of-the-art tools. Furthermore, *InsFinder* successfully detected 165 highly suspicious ISDs.

Acknowledgment

This research is supported in part by the National Key R&D Program of China (No. 2023YFB2703600), the NSFC-RGC Collaborative Research (No. 62461160332, CRS_HKUST602/24), the National Natural Science Foundation of China (No. 62032025), Guangdong Zhujiang Talent Program (No. 2023QN10X561), the Major Key Project of Peng Cheng Laboratory under Grant PCL2025AS07.

References

[1] W. Metcalfe *et al.*, "Ethereum, smart contracts, dapps," *Blockchain and Crypt Currency*, vol. 77, pp. 77–93, 2020.

[2] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–50, 2023.

[3] T. Min, H. Wang, Y. Guo, and W. Cai, "Blockchain games: A survey," in *2019 IEEE conference on games (CoG)*. IEEE, 2019, pp. 1–8.

[4] M. Bartoletti, J. H.-y. Chiang, and A. L. Lafuente, "Sok: lending pools in decentralized finance," in *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 2021, pp. 553–578.

[5] immunefi, "Hack analysis: Cream finance oct 2021," 2024. [Online]. Available: https://medium.com/immunefi/hack-analysis-cream-finance-oct-2021-fc222d913fc5

[6] "Decoding 220k read-only reentrancy exploit — quillaudits," 2022. [Online]. Available: https://quillaudits.medium.com/decoding-220k-read-only-reentrancy-exploit-quillaudits-30871d728ad5

[7] S. Chipolina, "Oracle exploit sees $89 million liquidated on compound," 2024. [Online]. Available: https://decrypt.co/49657/oracle-exploit-sees-100-million-liquidated-on-compound

[8] A. Ghaleb, J. Rubin, and K. Pattabiraman, "etainter: detecting gas-related vulnerabilities in smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 728–739.

[9] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 298–309.

[10] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Achecker: Statically detecting smart contract access control vulnerabilities," *Proc. ACM ICSE*, 2023.

[11] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.

[12] J. Zhang, Z. Zheng, Y. Nan, M. Ye, K. Ning, Y. Zhang, and W. Zhang, "Smartreco: Detecting read-only reentrancy via fine-grained cross-dapp analysis," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, 2025, pp. 1–12.

[13] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.

[14] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[15] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[16] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[17] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[18] DeFiHackLabs, "Defi hacks reproduce - foundry," 2024. [Online]. Available: https://github.com/SunWeb3Sec/DeFiHackLabs

[19] Immunefi, "Immunefi: Web3's leading bug bounty platform," 2024. [Online]. Available: https://immunefi.com/bug-bounty/

[20] C. Dannen, *Introducing Ethereum and solidity*. Springer, 2017, vol. 1.

[21] Vyper, "Vyper document," 2024. [Online]. Available: https://docs.vyperlang.org/en/latest/

[22] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[23] Z. Liao, S. Hao, Y. Nan, and Z. Zheng, "Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 980–991.

[24] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.

[25] PeckShield, "Value defi incident: Root cause analysis," 2024. [Online]. Available: https://peckshield.medium.com/value-defi-incident-root-cause-analysis-fbab71faf373
[26] Solidity, "Solidity source mappings - documentation," 2024. [Online]. Available: https://docs.soliditylang.org/en/latest/internals/source_mappings.html
[27] Z. Chu, J. Chen, Q. Chen, W. Yu, T. He, H. Wang, W. Peng, M. Liu, B. Qin, and T. Liu, "A survey of chain of thought reasoning: Advances, frontiers and future," *arXiv preprint arXiv:2309.15402*, 2023.
[28] Z. Zhong, Z. Zheng, H.-N. Dai, Q. Xue, J. Chen, and Y. Nan, "Prettysmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
[29] OpenAI, "Openai platform," 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o
[30] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.
[31] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.
[32] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.
[33] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," *arXiv preprint arXiv:2401.11817*, 2024.
[34] J. Li, J. Chen, R. Ren, X. Cheng, W. X. Zhao, J.-Y. Nie, and J.-R. Wen, "The dawn after the dark: An empirical study on factuality hallucination in large language models," *arXiv preprint arXiv:2401.03205*, 2024.
[35] H. Qiu, J. Huang, P. Gao, Q. Qi, X. Zhang, L. Shao, and S. Lu, "Longhalqa: Long-context hallucination evaluation for multimodal large language models," *arXiv preprint arXiv:2410.09962*, 2024.
[36] M. Ye, X. Lin, Y. Nan, J. Wu, and Z. Zheng, "Midas: Mining profitable exploits in on-chain smart contracts via feedback-driven fuzzing and differential analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 794–805.
[37] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 615–627.
[38] Linkedin, "44m usd hedgey finance exploit after auditing," 2025, https://www.linkedin.com/pulse/how-prevent-smart-contract-hacks-lessons-from-44m-hedgey-johnny-time-of22c.
[39] Olympix, "27m usd penpie exploit after auditing," 2025, https://www.olympix.ai/blog/penpie-exploit-case-study-leveraging-mutation-testing-to-prevent-smart-contract-vulnerabilities.
[40] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, "Nyx: Detecting exploitable front-running vulnerabilities in smart contracts," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 146–146.
[41] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.
[42] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.
[43] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
[44] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1176–1186.
[45] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 474–499, 2024.
[46] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *arXiv preprint arXiv:2405.02580*, 2024.
[47] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," *arXiv preprint arXiv:2308.11396*, 2023.